UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Biological Bayesian Optimizer

Autor: Pablo Díez del Pozo
Tutor: Eduardo César Garrido Merchán

junio 2020

Pablo Díez del Pozo
Biological Bayesian Optimizer

Pablo Díez del Pozo

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Agradecimientos

En primer lugar dar las gracias a Eduardo, por su paciencia y por guiarme en todo momento para que este trabajo llegara a buen puerto, he disfrutado mucho aprendiendo sobre cosas que no conocía.

Dar las gracias a mis padres, por ayudarme en todo momento, entenderme y remar siempre conmigo para conseguir los objetivos que me he ido planteando.

Y por ultimo, dar las gracias a Paula, por acompañarme siempre en este camino, unas veces más sencillo y otras más complicado, gracias por caminar siempre a mi lado ayudándome en todo momento.

Resumen

Los avances en el ámbito de la Inteligencia Artificial a lo largo de los últimos años han sido enormes, han aparecido nuevos tipos de redes y diariamente se encuentran nuevas aplicaciones en las que estos algoritmos consiguen muy buenos resultados. La *optimización bayesiana* es una técnica centrada en la optimización de funciones en las que se desconoce su expresión analítica, como podría ser aquella que forman todos los errores de predicción de un algoritmo, de los comentados anteriormente. En el presente trabajo se han combinado la *optimización bayesiana* y la optimización con *metaheurísticas* bioinspiradas. Se presenta una herramienta que en base a un problema planteado, bien sea una función o un algoritmo como una *red neuronal profunda* o de cualquier tipo, genera una recomendación de la que podría ser una metaheurística prometedora para realizar una optimización exhaustiva profunda futura. Para ello, combina ambas técnicas comentadas, un primer nivel de búsqueda en el que se aplican las *metaheurísticas* bioinspiradas y un segundo que optimiza los mejores resultados de estas. De esta forma, encontramos una *metaheurística* de la que se tiene la certeza que es la mejor que encuentra la *optimización bayesiana*.

La parte de las *metaheurísticas* se ha implementado utilizando pygmo2 y la *optimización bayesiana* con SMAC3. Además de la creación del programa, se han realizado una serie de experimentos para probar su funcionalidad y comparar si los resultados obtenidos son mejores que una búsqueda aleatoria en el mismo espacio de búsqueda.

Los experimentos consisten en 30 ejecuciones sobre un *perceptrón multicapa* y el algoritmo *XGBoost* en las que se obtiene finalmente una *metaheurística* optimizada por la herramienta, para cada uno de ellos. También se han realizado estos experimentos para la función matemática *Branin*, entre otras que no se muestran en el presente documento.

Los resultados no han sido los esperados, no se ha visto una diferencia relevante entre los resultados optimizados doblemente y las búsquedas aleatorias de los experimentos. Uno de los principales motivos, es la simplicidad del dataset utilizado para ellos, debido a las limitaciones computacionales. Se espera que con problemas más complicados, los resultados muestren una gran mejora entre los resultados con la herramienta y los obtenidos con una búsqueda aleatoria, queda como trabajo futuro.

Palabras clave

Optimización bayesiana, metaheurísticas

Abstract

Progress in Artificial Intelligence during last years had been huge, new *neural networks* had been developed and almost every day new uses are found for this kind algorithms. *Bayesian Optimization* it's an optimization technique with the aim of optimizing black box functions. Prediction errors of a *Machine Learning* algorithm shape a function of that type. In the present work, we combine *Bayesian Optimization* and bioinspired *metaheuristics* to build a tool to optimize a defined problem. That problem can be a mathematical function or a *Machine Learning* algorithm like *neural network*. The execution of the tool generates a promising *metaheuristic* that allows a future and exhaustive optmization. To do that, the tool has two levels of optimization, the first one optimizes the problem using *metaheuristics* and in the second one *bayesian optimization* is used to optimize the best results of the first level. That process guarantees that we found a *metaheuristic* that is certain to be the best one found by *Bayesian Optimization*.

Metaheuristics part, have been implemented using pygmo2, a *python* framework with a lot of bio-inspired metaheuristics. *Bayesian optimization* has been written using SMAC3. Also, we have coded a set of experiments to test the funtionality and compare if the results are better than doing an aleatory search in the same searching space.

Experiments consist in 30 executions of the program with *mulitlayer perceptron* and *XGBoost* as problems. The program returns, as a result, the best *metaheuristic* to optimize the problem in each of the executions. This experiments have been done optimizing *Branin* function, among others that are not presented in the document.

Examinating the results, there arent't much differences between the optimized values and the ones obtained using aleatory search. This fact can be explained with the simplicity of the dataset used to train and test *XGBoost* and *multilayer perceptron*, due to computacional limitations of the computer used to do the experiments. We expect that using a better machine for training and testing, with more complex datasets, will show much better results comparing the tool and *aleatory search*. It remains as future work.

Keywords

Bayesian optimization, metaheuristics

Índice

Listas

Lista de figuras

1

Introducción

La Inteligencia Artificial es una disciplina que está cobrando más importancia a medida que avanza el tiempo, los avances son continuos y las ayudas que supone en ámbito profesional y personal son cada vez más notables.

Por otro lado, la capacidad de cómputo con la que contamos actualmente es limitada, si la comparamos con la clase de problemas que se están intentando resolver, es cierto que se han resuelto problemas muy complicados que han supuesto grandes avances para la humanidad, pero aún quedan muchos otros sin resolver, en términos de computación clásica. Además, estos problemas son caros computacionalmente, lo que se traduce en la necesidad de invertir grandes cantidades de tiempo en entrenamiento de modelos y dinero para la compra de superordenadores o tarjetas gráficas.

Para resolver estos problemas se generan modelos, que pueden estar basados en estimación de probabilidades o redes neuronales, entre otros. Para conseguir que estos modelos sean útiles, es necesario realizar un entrenamiento de ellos y para que este entrenamiento sea suficiente para que los modelos sean robustos es necesaria una gran cantidad de datos. Actualmente, estos datos no es difícil obtenerlos, ya que vivimos en unos tiempos en los que la generación de información es constante.

Estos modelos tienen disponibles un gran número de posibilidades de configuración. En lenguaje técnico, estas posibilidades se conocen como *hiperparámetros* y permiten su adaptación a problemas de distinta índole que puedan plantearse. El ajuste de estos *hiperparámetros* al problema, representa uno de los mayores problemas existentes, ya que no hay ninguna manera de saber a priori cual es la mejor configuración.

Si aunamos las ideas de los dos últimos párrafos podemos llegar a la conclusión de que la estimación de *hiperparámetros* y entrenamiento del modelo con gran cantidad de datos, dan lugar a tiempos que se traducen en semanas o meses de entrenamiento, provocando el gasto de grandes cantidades de dinero en estos procesos.

En las siguientes líneas se presentarán los conceptos matemáticos aplicados y como han sido traducidos a código, de manera general. Estos conceptos son la *Optimización Bayesiana* y las *metaheurísticas*.

Lo primero es conocer el significado de *heurística*, que es un método utilizado en teoría de la computación para obtener soluciones cercanas a lo óptimo cuando no existe una solución que lo sea en las condiciones en las que nos encontremos. El término *metaheurística* es un método heurístico que permite obtener una solución satisfactoria cuando no existen algoritmos ni heurísticas que puedan encontrarla a un problema dado.

Otra parte fundamental del trabajo es la *optimización bayesiana*, de forma general, es una técnica utilizada para optimizar una función objetivo. Su uso se centra en la optimización de aquellas funciones con un gran coste de evaluación, ruido o aquellas en la que su expresión analítica es desconocida. Su objetivo es optimizar el valor de esas funciones en el menor número de evaluaciones posible, esto se consigue utilizando un modelo probabilístico y una función de adquisición, que permiten discernir que puntos son más prometedores para evaluar. Su uso es común en la optimización de *hiperparámetros* dentro del campo del *Aprendizaje Automático*.

Una vez presentado el problema con el que nos encontramos y los conceptos necesarios, es hora de definir el objetivo del trabajo, que es, de forma general, la programación de una herramienta que combine las *metaheurísticas* y la *optimización bayesiana* para optimizar algoritmos de *aprendizaje automático*. Más concretamente, lo que se pretende es encontrar una metaheurística candidata a optimizar algoritmos de *aprendizaje automático*, sabiendo que esa es la que mejores resultados ha obtenido, por la optimización de sus *hiperparámetros*, entre todas las metaheurísticas optimizadas con *optimización bayesiana*. Este concepto puede ser complicado de comprender, en esencia la herramienta optimizará el algoritmo utilizando *metaheurísticas* para posteriormente optimizar las mejores con *optimización bayesiana*.

Esta herramienta ofrece la solución al problema que presenta la gran cantidad de metaheurísticas disponibles que, además, pueden configurarse y no se tiene, a priori, ninguna garantía de que vayan a funcionar correctamente sobre el problema que se quiera resolver. En resumen, lo que se pretende con este trabajo es crear una herramienta que permita optimizar las técnicas de optimización y sugerir una de ellas como candidata a ser aquella que mejor optimice el problema, para una optimización futura y búsqueda de sus parámetros óptimos.

Aunque el objetivo final, es el expuesto anteriormente, para probar la herramienta también se han implementado optimizaciones dobles sobre funciones matemáticas, que quizá en algún momento también puedan ser de utilidad.

La programación ha sido realizada en *Python3.6.9* [2] en su totalidad, la *optimización bayesiana* ha sido codificada con el *framework* SMAC3 [3], que provee una implementación basada en *Random Forest* y las *metaheurísticas* han sido codificadas utilizando *Pygmo2* [4], otro *framework* que tiene implementadas un gran número de *metaheurísticas* bio-inspiradas.

Este documento representa una explicación de todo aquello que ha sido realizado a lo largo del *trabajo de fin de grado*. En primer lugar, se va a comentar el estado del arte de las *metaheurísticas*

y la optimización *hiperheurística* que se utilizan en el trabajo, seguidamente se explicará la definición del proyecto con sus objetivos, hipótesis, asunciones y restricciones definidas. Otra parte importante será la planificación que se ha llevado a cabo, en la que se incluye el diseño de la implementación, las tareas realizadas y una distribución de las mismas utilizando un *Diagrama de Gantt*, así como las horas que han sido necesarias para llevar a cabo cada una de ellas.

El siguiente punto tratado ha sido la implementación de la herramienta, que consta de una descripción del entorno, lenguajes de programación, librerías utilizadas, interacciones entre ellas y una descripción detallada de como se ha implementado cada parte definida en el modelo inicial. Los dos últimos apartados son los experimentos y las conclusiones, en ellos se muestran los resultados obtenidos y un análisis de los mismos, para ver si se han cumplido los objetivos e hipótesis inicialmente planteadas.

2

Estado del arte

Esta sección es una de las partes más importantes del documento, ya que se encargará de mostrar en que línea están yendo los estudios más actuales de los campos relacionados con el proyecto y que podemos aportar con él a ellos. Se va a dividir en tres partes, en la primera se detallará el estado del arte de las *metaheurísticas*, especialmente de aquellas que estén relacionadas con algoritmos *bioinspirados*, en la segunda una búsqueda de artículos relacionados con *optimización bayesiana* y por último, otro método usado para optimizar *hiperparámetros*, el *Tree-structured Parzen Estimator Approach(TPE)*.

2.1. Metaheurísticas

Una metaheurística, tal y como se ha explicado en la introducción del documento, es una forma de resolver un problema para el que no existe un algoritmo ni una heurística que resulten óptimas. Las metaheurísticas pueden ser muy variadas y en muchos casos se inspiran en el comportamiento de ciertos individuos que forman parte de la naturaleza. Este tipo es en el que nos vamos a centrar a lo largo del trabajo.

Se han encontrado estudios de *metaheurísticas* inspiradas en el comportamiento de los murciélagos [5], cuya base está centrada en el comportamiento de ecolocalización de estos animales, para implementarlo han intentado combinar las ventajas de otros algoritmos existentes con el comportamiento anteriormente explicado. Los resultado en simulaciones muestran que el rendimiento de este algoritmo parece muy superior al de otros que existen actualmente como pueden ser *particle swarm* o *algoritmos genéticos*.

Una de las metaheurísticas utilizadas, el algoritmo genético, tiene su aplicación también en el campo del *cloud computing* [6] o computación en la nube. En este trabajo se comenta como utilizando una combinación de algoritmos genéticos y otro basado en el comportamiento de las luciérnagas, se resuelven problemas NP-completos en tiempo cercanos al óptimo teórico.

2.2. Optimización bayesiana.

Se presenta a continuación el estado del arte de la *optimización bayesiana* como técnica para la optimización de *hiperparámetros*.

En esta publicación [7] se comenta el hecho de que la *optimización bayesiana* es una de las técnicas de moda para la optimización de funciones de caja negra, es decir, funciones de las que no se tiene conocimiento de su expresión analítica. Además, esta técnica es la más utilizada para la optimización de *hiper-parámetros* de funciones de *aprendizaje automático* en base a la minimización de su error. A pesar de esto, también requiere de una amplia configuración, que si no es la correcta conduce a malos resultados. En este trabajo se propone un trabajo sobre *optimización bayesiana* automática utilizando distintas heurísticas que automáticamente configuran la función de adquisición.

En el siguiente artículo [8] muestra la capacidad de la optimización bayesiana para estimar hiperparámetros que conduzcan a soluciones óptimas de problemas tan importantes como CARP (Capacitated Arc Routing Problem), uno de los mayores retos del sistema logístico. Esta estimación se solía realizar de manera empírica y en la publicación se el uso del algoritmo SAHiD pero con la configuración de parámetros generada a partir de *Optimización Bayesiana*. Los resultados mostraron una gran mejora de rendimiento con la versión optimizada frente al algoritmo sin utilizar.

La *optimización bayesiana* se aplica también a otras ramas de la ciencia, en el siguiente artículo [9] se hace una comparación del uso de un algoritmo de *optimización bayesiana* frente a búsqueda aleatoria uniforme en el campo de la química. Los métodos de generación de confórmeros se basan en la generación de confórmeros con diversidad geométrica, en lugar de utilizar probabilidad para buscar el más probable entre ellos o el que tenga la mínima energía. La aplicación de optimización bayesiana se centra en la minimización energética de los mismos. En este estudio concluyen que la aplicación de esta técnica requiere muchas menos evaluaciones que la búsqueda aleatoria uniforme para encontrar el mínimo de la energía, más concretamente se ahorra entre un 20 y un 40 %.

En la industria farmacéutica también se han probado las capacidades de la *optimización bayesiana* [10] para la eliminación de experimentos innecesarios y la aceleración de los métodos de desarrollo utilizados. En el experimento se optimizaron la formulación y el procesado de parámetros de pastillas de disolución oral obteniendo como resultado que la *optimización bayesiana* reducía de forma muy eficiente el número de experimentos para la obtención de la formulación y parámetros óptimos frente a una búsqueda aleatoria.

2.3. TPE (tree-structured Parzen Estimator Approach)

Otro de los métodos más utilizados para la optimización de *hiperparámetros* son los *TPE*. En este contexto, existe un estudio [11] que ha desarrollado un modelo para la administración de pronósticos y

salud basado en una *LSTM*, un tipo de red neuronal profunda, para tratar con problemas relacionados con dependencias de series temporales. Además de este modelo, para estimar los *hiperparámetros* utilizan el un algoritmo de Optimización Bayesiana llamado *Tree-sctructured Parzen Estimator (TPE)*.

Otro grupo de investigadores han estimado los hiper-parámetros de una CMA-ES [12], un algoritmo muy utilizado para la optimización de funciones no lineales y no convexas. Este algoritmo cuenta con tres parámetros c_c, c_1 y c_μ que son muy importantes para las actualizaciones de la matriz de covarianza. Utilizando *TPE*, se modelan la distribución de calidad de la implementación y la distribución condicional de la distribución una vez medida la solución. Se comparan el *CMA-ES* sin optimizar y el optimizado, consiguiendo una convergencia mucho más rápida y óptima al usarse *TPE* para la optimización de las distribuciones.

Las herramientas utilizadas en el trabajo son de uso común y muy importantes en los campos de la optimización, ya que existen un gran número de estudios que muestran las capacidades de la *optimización bayesiana* aplicada a hiper-parámetros de algoritmos y también de distintas metaheurísticas *bioinspiradas* que permiten resolver problemas de optimización. Por tanto, aunando estas dos ideas se pretende crear una herramienta funcional que permita la optimización de *hiperparámetros* de cualquier tipo de algoritmo que se desee.

3

Definición del proyecto

En este capítulo se va a exponer una lista con todos los objetivos planteados para el trabajo, así como, las asunciones necesarias y por último, las restricciones para poder llevarlo a cabo en el ámbito educativo y que su extensión se adapte también a las capacidades con las que se cuenta.

3.1. Objetivos

En este apartado se definen los objetivos planteados al inicio del trabajo. En general la definición de objetivos es una buena manera de poder extraer conclusiones una vez se han hecho experimentos con la implementación propuesta y también de tener un objetivo claro a la hora de desarrollar, conociendo perfectamente a donde se tiene que llegar con la herramienta propuesta. Estos objetivos son los siguientes:

o.1: desarrollar una herramienta que permita la optimización de hiper-parámetros de algoritmos de *Aprendizaje Automático* utilizando una búsqueda de dos niveles, el primero utilizando *metaheurísticas* y el segundo utilizando *Optimización Bayesiana*.

o.2: comparar la búsqueda en doble nivel implementada frente a la utilización de una única metaheurística en problemas definidos y una búsqueda aleatoria en el mismo espacio definido para la búsqueda de doble nivel.

o.3: obtención de una recomendación de metaheurística con sus *hiperparámetros*, que sirva como aproximación a un problema concreto planteado y permita en un futuro hacer una optimización más exhaustiva del problema utilizándola.

3.2. Asunciones

En el apartado de asunciones se quieren destacar tres conceptos importantes:

a.1: se asume que la función de error perteneciente al espacio imagen de los problemas de optimi-

zación está contenida en el espacio funcional definido por la media y la varianza del modelo probabilístico empleado.

a.2: se asume que con una mayor capacidad de cómputo se podrían obtener resultados que puedan destacar más la calidad de las soluciones obtenidas por la herramienta. Dada la capacidad de cómputo actual representan, únicamente, una aproximación muy limitada de los problemas planteados.

a.3: se asume que las *metaheurísticas* empleadas son un resumen de todas aquellas publicadas y podría extenderse su funcionamiento a cualquier otra de elección.

3.3. Restricciones

En este apartado se exponen las restricciones que se han considerado a la hora de desarrollar el trabajo:

r.1: dado que el proyecto no cuenta con financiación, los experimentos llevados a cabo han sido realizados en un *Intel Core i5-3570K*, con 16GB de memoria *RAM* y sin gráfica dedicada, lo que ha provocado no hacer más experimentos y de mayor calidad.

r.2: el tiempo para llevar a cabo este proyecto, en este caso, ha sido de aproximadamente un año y no ha sido a tiempo completo. Esto es una limitación ya que con más tiempo se podrían haber explorado más algoritmos y podido llegar a resultados más interesantes.

4

Diseño del proyecto

4.1. Diseño del modelo

En este apartado se va a explicar el funcionamiento, desde un punto de vista matemático, de la *optimización bayesiana* que se está utilizando en el trabajo.

Todos los algoritmos de aprendizaje automático cuentan con un gran número de *hiperparámetros*, cuyos valores hacen que un modelo tenga mejores o peores resultados. Habitualmente, el ajuste de estos *hiperparámetros* suele ser hecho por alguna persona experta pero, su conocimiento, no será generalizable, ya que existen un gran número de problemas que van creciendo día a día y no es posible tener conocimientos de todos ellos.

Una posible solución a este problema sería utilizar búsqueda uniforme o de rejilla para obtener buenos valores. Esta solución tiene un problema, a mayor dimensión del espacio de parámetros, más complicado se vuelve hacer funcionar a estas técnicas correctamente.

Como alternativa a este proceso surge la *optimización bayesiana*, tan comentada a lo largo del trabajo, que se va a explicar formalmente en las siguientes lineas. En primer lugar, es necesario asumir que la función generada por el error de una predicción es suave, que quiere decir que sus derivadas de cualquier orden son continuas. En concreto, esta técnica es especialmente útil en la optimización de *funciones de caja negra*, que son funciones de las que no se conoce su expresión analítica, su evaluación es costosa y son ruidosas, lo que concuerda con la función que generan los errores de predicción de un algoritmo de *aprendizaje automático* partiendo de un dataset concreto aleatorizado.

Partimos de una función de evaluación de caga negra $f(.)$, a la que se le añade ruido de la forma $y_i = f(x_i) + \epsilon_i$ donde ϵ_i es el ruido añadido a la evaluación de la función inicial.

La *optimización bayesiana* es un método muy bueno para reducir el número de evaluaciones necesarias para optimizar $f(.)$. En cada iteración $t = 1, 2, 3...$ del proceso de optimización se va generando un modelo probabilístico, en nuestro caso un *random forest*. Para ajustar el modelo se toman las observaciones de la función objetivo $\{y_i\}_{y=1}^{t-1}$. Esta incertidumbre generada por el *random forest* permite construir una función de adquisición $\alpha(.)$. Esta función toma un valor, que depende de la entrada que

reciba y representa el valor de utilidad que va a tener la función de evaluación $f(.)$.

El algoritmo seguirá entrenando el modelo probabilístico, dando valores a la función de adquisición $\alpha(.)$ hasta que su valor sea máximo, en ese punto es cuando se evaluará $f(.)$, por tanto, el valor de $\alpha(.)$ solo depende del modelo.

La clave de la *optimización bayesiana* es que la evaluación de $\alpha(.)$ es poco costosa si la comparamos con $f(.)$, ya que la segunda de ellas representa el reentrenamiento del modelo completo. Esto es así porque, como se ha comentado antes, la función de adquisición únicamente depende de la distribución que predice el *random forest* para $f(.)$, en un punto candidato para ser explorado ***x***. Si los datos observados por el algoritmo hasta el instante *t - 1* son $D_i = \{x_i, y_i\}_{y=1}^{t-1}$. La distribución de probabilidad para $f(.)$ viene dada por la media $\mu(\mathbf{x})$ y la varianza $\sigma^2(\mathbf{x})$. Para obtener estos valores sería necesario tener la función de covarianza $k(.,.)$.

Existen distintos tipos de funciones de adquisición entre las que destacan *expected improvement* o *probability of improvement*.

Como ha podido observarse, la *optimización bayesiana* es un método que se encarga de elegir los puntos a evaluar de manera concienzuda, intentando evaluar en todo momento aquellos puntos más prometedores. Esto nos lleva a la conclusión de que se genera un balance entre exploración del espacio de búsqueda y su explotación, en contraposición con la búsqueda aleatoria o uniforme, cuyo fundamento principal es la exploración completa del espacio de búsqueda.

Algorithm 1 Bayesian optimization of a black-box objective function

Result: Optimize the mean of the Random Forest to find the solution.

1: **for** t = 1, 2, 3, ... max steps **do**
2: Find the next point to evaluate by optimizing the acquisition function: $\mathbf{x}_t = \arg\max_x \alpha(\mathbf{x}|D_{1:t-1})$.
3: Evaluate the black-box objetive $f(.)$ at $\mathbf{x}_t$: $y_t = f(x_t) + \epsilon_t$.
4: Augment the observed data $D_{1:t} = D_{1:t-1} \cup \mathbf{x}_t, y_t$.
5: Update the Random Forest model using $D_{1:t}$.
6: **end for**=0

Figura 4.1: Algoritmo de *optimización bayesiana* seguido por la librería *SMAC3* y utilizado en el trabajo. Adaptado de [1].

En la figura 4.1 se muestra el algoritmo que sigue la *optimización bayesiana* explicado en las lineas anteriores.

4.2. Diseño técnico

El diseño técnico de la aplicación se ha realizado utilizando un *Diagrama de Flujo de Datos* [13], más conocido como *DFD*. Este modelo permite representar el proceso que sigue la información que utiliza el sistema, desde el inicio hasta el final, es decir, los datos de los que se parten hasta obtener

la salida deseada, para ello se utilizan entidades externas, de las que no hacemos uso en la implementación, procesos, que es aquello que se encarga de modificar los datos de entrada, almacenes de datos que es donde la información que genera el software se almacena y por último, flujos de datos que relacionan los elementos anteriores permitiendo describir como viaja la información entre esos procesos, almacenes de datos y unidades externas definidas.

Existen 3 niveles de abstracción a la hora de realizar un *DFD*, el nivel 0, donde se muestran las relaciones entre los procesos y las unidades externas. Su objetivo es mostrar como interacciona el programa con los elementos externos que le envían o recogen información. En el caso que estamos viendo, no contamos con unidades externas con las que interaccionemos, por lo tanto, este nivel no es necesario. El nivel 1 es una descripción de los principales procesos que modifican la información de entrada para producir la salida y como se relacionan entre si, de manera global, sin entrar en detalle. El nivel 2 representa una descomposición exhaustiva de todos los procesos y como interaccionan y modifican la información internamente.

Una vez analizado cómo funciona un *DFD*, nuestra aplicación necesita uno de nivel 1 que describa el proceso general que siguen los datos de entrada, como pasan por las *metaheurísticas* y la *optimización bayesiana* produciendo el valor optimizado en el caso de las funciones y el valor optimizado con los *hiperparámetros*, que se ha obtenido de la función de *Aprendizaje Automático* utilizada.

Además del diagrama general, se ha desarrollado uno concreto para explicar como crear un problema de optimización que ejecute un algoritmo de *Aprendizaje Automático*, no es exactamente un *DFD* ya que con los componentes que nos provee no podríamos expresar correctamente la funcionalidad. Por otro lado, el programa diseñado tampoco es un programa al uso, por ello, se cree conveniente esta variación en la metodología para expresar ciertas funcionalidades.

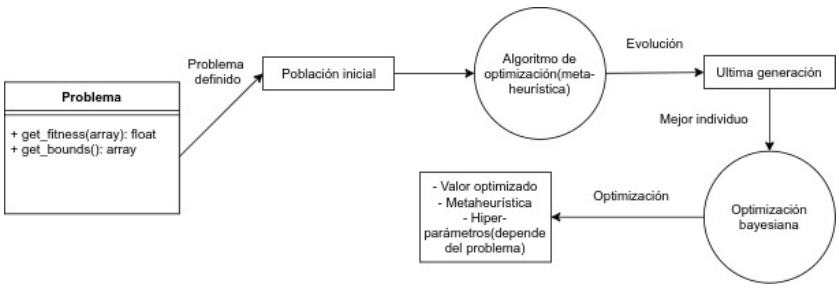

Figura 4.2: Diagrama de Flujo de Datos general.

En la figura 4.2 se pueden observar dos procesos, el *Algoritmo de Optimización* y la *optimización bayesiana*, que representan los dos niveles de optimización propuestos en este trabajo.

Es necesario saber que las *metaheurísticas*, tal y como las implementa pygmo2, necesitan tener una población de un número definido de individuos, sobre la que se producirá la optimización, evolucionándola hasta llegar a una solución final, que representa la óptima al problema planteado. La salida generada por el programa la representa el almacén de datos que genera la *Optimización Bayesiana* y

la población inicial el que se encuentra como entrada del *Algoritmo de Optimización.*

Los distintos problemas de optimización que se le proponen al *Algoritmo de Optimización*, en pygmo2 se implementan como una clase, que define los dos métodos que aparecen en ella y permiten definir el problema a optimizar, mientras que el rango de valores que pueden tomar las variables dependientes de las funciones las maneja *get_bounds()*.

De forma general, la *Optimización Bayesiana* se encarga de optimizar los valores previamente optimizados, valga la redundancia, por las *metaheurísticas*, generando un valor final, doblemente optimizado, que representa el mejor valor obtenido y con que *metaheurística* se obtiene.

Cuando se quiere optimizar una función, el problema definido es sencillo, únicamente son necesarios la función y los límites inferiores y superiores que van a tomar las variables dependientes, pero, para optimizar *hiperparámetros* la situación se complica, ya que queremos que esos límites definan los valores que toman esos *hiperparámetros* y por tanto, el problema tiene que definir un algoritmo de *Aprendizaje Automático*, en el caso de este trabajo, *XGBoost*, un *Perceptrón Multicapa* muy simple y una *Red Neuronal Convolucional*.

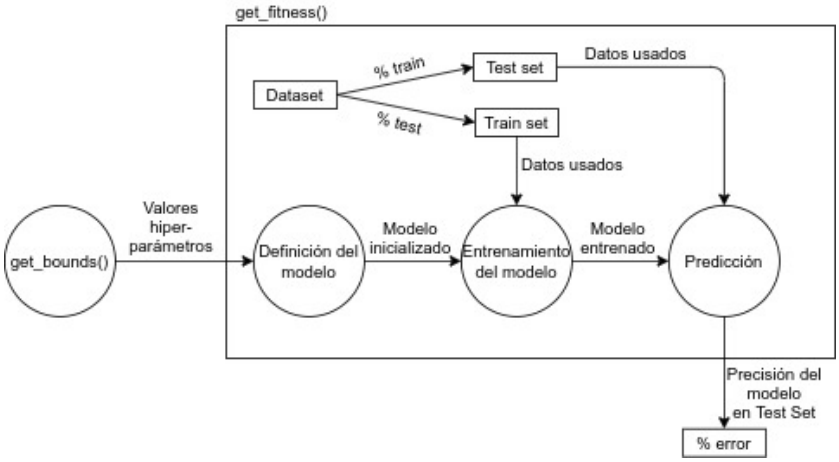

Figura 4.3: Diagrama de Flujo de Datos de un problema de *Aprendizaje Automático*.

En la figura 4.3 se muestra como se implementa un problema de este tipo en pygmo2. Como se ha comentado antes es una mezcla, de un *DFD* y alguna componente propia que creemos que clarifica el proceso que siguen. El recuadro grande muestra el funcionamiento de *get_fitness()*. Se utiliza un *dataset*, que se subdivide en dos particiones, la de *train* y la de *test*, en base a un porcentaje definido. Lo siguiente es definir el modelo, esto implica inicializar los *hiperparámetros*, estos no tomaran un valor fijo si no que irán siendo estimados en función de un rango de valores, definido en la función *get_bounds()*. Por último, se entrenará el modelo con la partición de *train* y se generará una predicción con ese modelo ya entrenado utilizando los datos de *test*, que servirá para calcular un porcentaje de error, que será el resultado devuelto por la función. Este proceso será repetido por la *metaheurística* en las *n* generaciones definidas obteniendo finalmente el mejor individuo, con el menor error y los

valores que han tomado los *hiperparámetros* para obtenerlo. Este proceso se hará un número de iteraciones, definido por la *Optimización Bayesiana* que acabará devolviendo la mejor *metaheurística* que optimiza el problema, el porcentaje de error y los *hiperparámetros* que lo obtiene, llegando al resultado deseado.

4.3. Planificación del proyecto

En este apartado se va a detallar como se ha llevado a cabo la planificación del Trabajo de Fin de Grado. Para ello, se listarán todas las tareas que se planificaron en el inicio y como han sido distribuidas a lo largo del tiempo de duración del trabajo. Para ello, se utilizará un Diagrama de *Gantt*, ya que permite una visualización rápida y clara de la planificación seguida. No reflejará relaciones entre las tareas, porque en este caso no es demasiado relevante, ya que se definieron de manera secuencial y con la menor relación posible entre ellas para evitar problemas de solapamiento y tener una estructura iterativa y bien marcada de las fases desde el inicio.

4.3.1. Tareas definidas

A continuación se enumeran todas las tareas en las que se ha subdivido el proyecto:

- Tarea 1: Investigación, estudio *metaheurísticas* y análisis de aquellas que se van a utilizar.
- Tarea 2: Investigación y estudio del funcionamiento de la *Optimización Bayesiana* desde un punto de vista matemático.
- Tarea 3: Estudio del problema que se plantea y diseño de una solución de alto nivel que lo resuelva. Esta fase o tarea es en la que se desarrolló en los diagramas y explicaciones que aparecen en la sección 4.2.
- Tarea 4: Implementación de ejemplos de meta-heurísticas optimizando la función de Booth utilizando la librería. pygmo2 de python.
- Tarea 5: Implementación de ejemplos de Optimización Bayesiana sobre la función de Booth, con el objetivo de comprender el funcionamiento de la librería de python SMAC3.
- Tarea 6: Pruebas sobre las implementaciones de las Tareas 4 y 5, modificando hiper-parámetros para comprender el funcionamiento.
- Tarea 7: Implementación de una búsqueda en dos niveles sobre la función de Booth:
  - El primer nivel es una optimización del valor mínimo utilizando *metaheurísticas*.
  - El segundo nivel es la *Optimización Bayesiana* de los valores optimizados de las *metaheurísticas* obtenidas por el nivel 1.
- Tarea 8: Pruebas de distintas configuraciones de las *metaheurísticas* y de la *optimización bayesiana* y su funcionamiento conjunto sobre la implementación de la tarea 7.
- Tarea 9: Ampliación de las funciones matemáticas que se optimizan en la búsqueda de doble nivel. Entre ellas se encuentran *Griewank, Rastrigin o Schwefel*, entre otras que serán explicadas en secciones del documento. Estas nuevas funciones son más complicadas de optimizar que la función de *Booth*.
- Tarea 10: Optimización de *hiperparámetros* del algoritmo de aprendizaje automático *XGBoost*.

- Tarea 11: Pruebas de la optimización de *XGBoost*.
- Tarea 12: Optimización de *hiperparámetros* de una *Red Neuronal Convolucional*.
- Tarea 13: Pruebas de la optimización de la *Red Neuronal Convolucional*.
- Tarea 14: Experimentos de ejecución de la optimización en dos niveles de funciones matemáticas.
- Tarea 15: Experimento de optimización de los *hiperparámetros* de *XGBoost*.
- Tarea 16: Experimento de optimización de los *hiperparámetros* que componen una *Red Neuronal Convolucional*. Esta tarea, fue necesario suprimirla, ya que no se contaba con la capacidad de computo suficiente, se explicará más en detalle en la sección 4.3.2.
- Tarea 17: realización de búsquedas aleatorias, sin *optimización bayesiana* sobre *XGBoost* y la *Red Neuronal Convolucional*, con el objetivo de evaluar los resultados de los experimentos.
- Tarea 18: Optimización de *hiperparámetros* de un *perceptrón multicapa*.
- Tarea 19: Pruebas de la optimización del *perceptron multicapa*
- Tarea 20: Experimento de optimización de los *hiperparámetros* del *perceptrón multicapa.*
- Tarea 21: Evaluación de los resultados obtenidos en los experimentos.
- Tarea 22: Redacción de la memoria final del Trabajo Final de Grado.
- Tarea 23: Elaboración de *PowerPoint* y práctica para la presentación del proyecto.

Estas tareas representan una subdivisión del problema planteado inicialmente, de tal forma que es mucho más abordable e iterativo, ya que es un concepto bastante amplio y complejo para ser tratado de forma global en primera instancia. Serán explicadas profundamente en el capítulo 5.

Las tarea 6 es la fundamental dentro del programa implementado, ya que representa la funcionalidad principal de la aplicación, su diseño está recogido en la sección 4.2 y equivaldría a la figura 4.2. La otra funcionalidad importante es la que recogen las tareas de la 10 a la 20 y se puede ver el diseño principal de la problemática que genera en la figura 4.3.

4.3.2. Diagrama de Gantt

En esta sección se va a presentar el Diagrama de Gantt generado para la distribución de tareas a lo largo del tiempo de duración del Trabajo de Fin de Grado. Las tareas que lo componen están definidas en el apartado 4.3.1.

En primer lugar, podemos observar en la tabla 4.1 como fue el reparto final de tareas y su división temporal, además la última columna indica la cantidad de horas que cada tarea a requerido, que permite comprobar si la estimación fue correcta.

En la figura 4.4 se muestra la duración temporal de las distintas tareas descritas en el apartado 4.3.1, los números que hay en cada una de ellas, dentro del diagrama, son el número de horas invertidas hasta su finalización, que es próxima a las 360 horas de trabajo que se estiman para el Trabajo de Fin de Grado. No se han definido relaciones entre las tareas, ya que, todas ellas son secuenciales, exceptuando la redacción de la memoria y los experimentos, que siempre empiezan cuando la parte

Tarea	Fecha Inicio	Fecha Final	Horas invertidas
Tarea 1	01/07/2019	01/08/2019	14
Tarea 2	15/08/2019	15/09/2019	26
Tarea 3	18/09/2019	01/10/2019	26
Tarea 4	02/10/2019	10/10/2019	10
Tarea 5	11/10/2019	20/10/2019	20
Tarea 6	21/10/2019	25/10/2019	10
Tarea 7	26/10/2019	15/11/2019	30
Tarea 8	16/11/2019	21/11/2019	8
Tarea 9	22/11/2019	28/11/2019	10
Tarea 10	23/12/2019	31/12/2019	16
Tarea 11	02/01/2020	06/01/2020	8
Tarea 12	07/01/2020	15/01/2020	30
Tarea 13	16/01/2020	18/01/2020	6
Tarea 14	01/12/2019	15/12/2019	4
Tarea 15	07/01/2020	25/01/2020	6
Tarea 16	19/01/2020	06/03/2020	22
Tarea 17	01/03/2020	01/04/2020	10
Tarea 18	10/03/2020	13/03/2020	8
Tarea 19	14/03/2020	16/03/2020	4
Tarea 20	17/03/2020	31/03/2020	12
Tarea 21	01/04/2020	15/04/2020	20
Tarea 22	15/04/2020	20/05/2020	80
Tarea 23	01/06/2020	10/06/2020	20

Tabla 4.1: Tabla que representa el reparto de tareas y las horas invertidas en ellas.

de código correspondiente a él ha sido terminada. La fecha de inicio fue el 1 de Julio de 2019 y la fecha de finalización el 10 de Junio de 2020, incluyendo el tiempo para preparar y practicar la presentación.

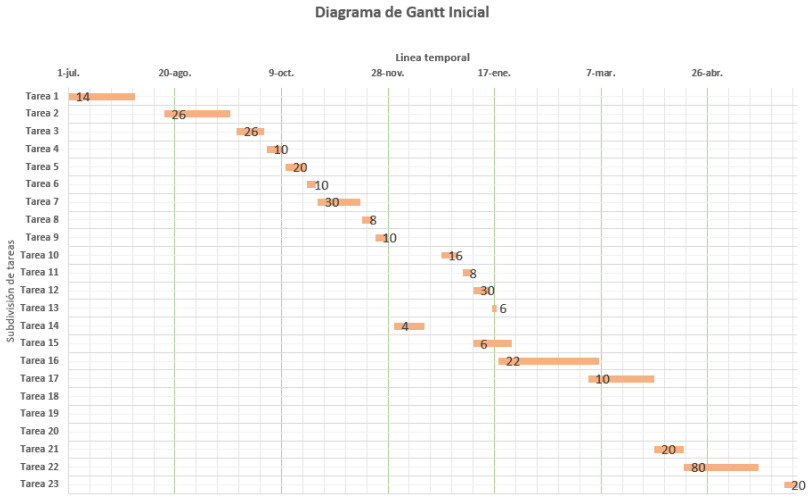

Figura 4.4: Diagrama de Gantt.

Esta fue la planificación inicial que se planteo, pero, durante la ejecución del experimento de la Red Neuronal Convolucional se llegó a la conclusión de que era muy caro computacionalmente y no teníamos capacidad suficiente para realizarlo. Por tanto, se añadieron nuevas tareas para implementar la optimización sobre un *Perceptrón Multicapa*, que no requiere tanta potencia. Estos cambios se reflejan en la figura 4.5.

En primera instancia se querían fijar las bases de conocimiento teórico y práctico de las herramientas, tanto matemáticas como de programación, y que se han usado para obtener una solución al problema planteado. Una vez completada esta fase, se desarrolló un diseño de alto nivel de la herra-

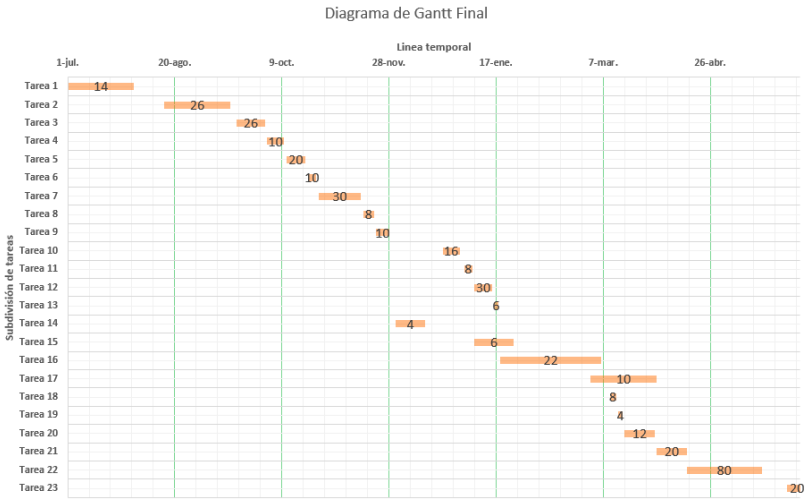

Figura 4.5: Diagrama de Gantt actualizado.

mienta a implementar, los detalles están en la sección 4.2.

En Diciembre de 2019 se comenzó a implementar ejemplos de uso de las librerías, pero por separado, para comprender el funcionamiento interno tanto de pygmo2 como de SMAC3. Una vez probados e implementados, se comenzó a desarrollar la optimización en dos niveles propuesta pero restringida a optimizar funciones matemáticas. Con esto hecho ya se tenia la base para el objetivo fundamental del trabajo, optimizar *hiperparámetros*.

Decidimos optimizar el algoritmo *XGBoost* y una *Red Neuronal Convolucional* que, como hemos comentado anteriormente, se sustituyó por un *Perceptrón Multicapa*.

Una vez contábamos con la implementación final, comenzó la fase de experimentos, en ella se ejecutaron baterías de pruebas que nos permitieran saber si realmente se había encontrado una solución para el problema válida. Para saberlo, también fue necesario ejecutar búsquedas aleatorias y comparar los resultados con los generados por los experimentos y llegar a una conclusión final. Este proceso estará detallado en el capítulo 6.

Los experimentos son lo único que se ha realizado en paralelo, ya que, en general, son caros computacionalmente y era mejor idea seguir avanzando que esperar a que terminaran por completo. También se comenzó a escribir la memoria antes de terminar esos experimentos por completo, se consideró una buena idea para aligerar la carga al final de cuatrimestre donde siempre se tiene menos tiempo, a pesar de que se vio incrementada por los problemas ya comentados.

La duración temporal no está relacionada con el número de horas invertidas en ciertas tareas, en el caso de los experimentos porque no se tiene en cuenta el tiempo de ejecución de los mismos, únicamente su codificación e interpretación. En la tabla 4.1 se muestra una descomposición del número de horas y la duración temporal estimada para cada tarea que clarifica todo lo explicado en este apartado.

En general, algunas tareas se sobreestimaron en un principio, dedicándole en la práctica menos tiempo del que se pensaba inicialmente y a otras lo contrario. La descomposición y planificación de las tareas únicamente ha reportado beneficios, permitiendo incorporar dos nuevas tareas bastante cerca de la fecha fijada de finalización, que no han supuesto una gran carga de trabajo extra y han resultado ser una gran y rápida solución al problema que se había planteado y en un primer momento no contemplamos.

5

Implementación

En este capítulo se van a dar detalles de todo el proceso de instalación de librerías necesarias, lenguajes y herramientas de programación utilizadas y como se implementan todos los elementos descritos en la sección 4.2 profundamente.

5.1. Lenguajes y herramientas

Como lenguaje principal de la implementación se ha utilizado *Python3*, un lenguaje interpretado, orientado a objetos, de alto nivel y con tipado dinámico. Es un lenguaje muy potente y utilizado en la actualidad.

Se ha elegido este lenguaje por estar muy extendido en la comunidad de la *Inteligencia Artificial*, por su simplicidad para escribir y leer código y por último, porque se contaba con una gran base programando con él, por tanto, la tarea del *Trabajo de Fin de Grado* era únicamente familiarizarse con los conceptos a aplicar y las librerías a utilizar y no conocer el funcionamiento del lenguaje, lo que permitía centrarse en la comprensión, diseño e implementación del problema.

Todas las fases del proyecto han sido llevadas a cabo en *Ubuntu*, más concretamente en su versión *18.04*. Es un sistema operativo de código abierto, basado en GNU/Linux. En primer lugar, se conocía perfectamente su funcionamiento y en él se tenía soporte para todas las librerías a utilizar y para *Python3.6.9*.

Estas han sido las herramientas principales, pero no las únicas. Para el desarrollo se ha utilizado *Visual Studio Code*, conocido editor de texto desarrollado por *Microsoft*. Tanto pygmo2 como SMAC3 tienen soporte por parte del instalador de paquetes *pip3*, que permite instalar todo tipo de paquetes soportados. Estas librerías tendrán un apartado propio donde se explica el uso necesario para llevar a cabo la solución al problema planteado.

Además de estas herramientas, se ha utilizado *bash*, como lenguaje de *scripting* para generar los distintos *benchmarks* utilizados en los experimentos.

5.2. Preparación del entorno de ejecución

Como se ha comentado en la sección 5.1 hemos utilizado *Python3.6.9* como lenguaje y *Ubuntu 18.04* como sistema operativo para el desarrollo. En las siguientes lineas se detallará todo lo necesario para desplegar el entorno de ejecución, comando incluidos, que permita poder hacer un uso completo de la solución implementada.

En primer lugar, será necesario tener la versión de *Python3.6.9* mínimo, se ha desarrollado utilizando esta, pero debería funcionar para versiones más actualizadas de la misma forma. Otro requisito imprescindible es instalar *pip3*, el gestor de paquetes de *Python3*.

Para poder trabajar, es necesario un entorno donde únicamente se tengan instalados aquellos paquetes imprescindibles para el funcionamiento del programa, se utilizará el programa *Virtualenv*. Esto es bastante recomendable, ya que, las versiones de algunos paquetes no son las más actualizadas y podrían aparecer conflictos si utilizamos estos paquetes en versiones actuales para otros propósitos. Por tanto, esta es la mejor solución.

Las librerías que vamos a utilizar *Pygmo2* y *SMAC3*, son librerías de optimización y su base está escrita en *C++*, para ello es necesario tener su compilador instalado. Además, dada su necesaria velocidad de ejecución es necesario instalar la librería *Boost* en Linux. Por último, es necesario instalar una última librería, swig, que permite conectar esas implementaciones en *C++* a *Python3*.

Haciendo uso del gestor de paquetes de *Python3* se instalará *smac* y *pygmo2*. Al ejecutar sus instalaciones también lo harán todas sus dependencias. Con todo lo explicado tendríamos un sistema capaz de ejecutar sin ninguna restricción el programa implementado.

5.3. Librerías utilizadas

En este apartado se va a presentar el funcionamiento de las librerías más relevantes en el desarrollo de la solución, en la comprensión de su funcionamiento y uso se han centrado gran parte de los esfuerzos del trabajo. Se irá mostrando como su utilización resuelve cada una de las partes descritas en la sección 4.2 y las interacciones que aparecen entre ellas en el subapartado final.

En este apartado también se explicará la implementación seguida para definir como problemas de búsqueda tanto las funciones matemáticas como *XGBoost*, la *Red Neuronal Convolucion* y el *Perceptrón Multicapa*.

Por último, para presentar los resultados fue necesario realizar gráficas que mostraran su evolución a lo largo de distintas pruebas. Para ello se ha utilizado la librería *matplotlib*, la cuál también se conocía y resulto sencillo generar las distintas gráficas para presentar de forma correcta los resultados. Esta librería es de uso general, por ello, no se ha considerado necesario dedicarle un apartado especial.

5.3.1. Pygmo2

La librería *Pygmo2* [4] es una librería científica que permite la optimización paralela masiva. Su objetivo principal es ofrecer al usuario una interfaz de algoritmos y problemas de optimización unificada y facilitar de entornos con gran paralización de forma sencilla. Utiliza algoritmos de optimización bio-inspirados.

Este *framework* se utiliza para implementar la búsqueda de primer nivel que podemos observar en el *DFD* como *Algoritmo de Optimización*. Para ello se definen cuatro *metaheurísticas* provistas por *Pygmo2*, que se aplican sobre un espacio de búsqueda definido.

Representación de un problema de optimización

En primer lugar es necesario explicar como se codifica el problema de optimización. Podemos ver un ejemplo de código de la definición de la función de *Booth* en la figura A.6. Las dos funciones imprescindibles a implementar son *fitness()* y *get_bounds()*, aunque hay muchas más, en función de lo que se quiera optimizar y como. La función fitness(), es la que recoge la funcionalidad principal del problema, en este caso se encuentra definida la función de Booth $f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2)^2$.

Tenemos dos variables dependientes, x_1 y x_2, en *Pygmo2* se representan como coordenadas del vector que recibe la función *fitness()* como argumento. El objetivo de un algoritmo de optimización es minimizar el valor, por tanto, en este caso, querremos obtener el mínimo global de la función de *Booth*. Este valor depende de los valores que tomen las variables dependientes y a priori, no se conocen. Para poder definir el rango de valores que toman las variables de la función se utiliza la función *getbounds()*, la primera lista representa el valor máximo y la segunda el valor mínimo que va a tomar.

En estos problemas, se está dando valor a las variables dependientes de la función matemática pero, el objetivo principal del proyecto es optimizar los *hiperparámetros* de algoritmos de *Aprendizaje Automático* para simplificar esta ardua tarea. Esto se realiza siguiendo el mismo proceso que con una función matemática, lo único que esas variables dependientes pasan a a ser aquellos *hiperparámetros* que queramos optimizar. En la figura A.7 se ejemplifica lo expuesto anteriormente.

Este proceso es el mismo tanto para la *Red Neuronal Convolucional* como en el *Perceptrón Multicapa*. Tanto la implementación de *XGBoost* como de los algoritmos anteriormente mencionados será comentada en profundidad en las siguientes secciones del documento, en esta únicamente se muestra de forma general como funciona la clase *Problema* de *Pygmo2*.

Por último, es necesario explicar un pequeño concepto para comprender la implementación. En algoritmos *bioinspirados*, los individuos están compuestos por un conjunto de cromosomas, que representan una posible solución al problema que se le plantee. En nuestro caso, al optimizar funciones, estos cromosomas serán las variables dependientes y al optimizar algoritmos de *Aprendizaje Automático* serán aquellos *hiperparámetros* que queramos optimizar. Por último, para decidir que individuos son

mejores es necesario darles una puntuación, que se obtiene de la función *fitness*, para las funciones el valor de la misma y los algoritmos el porcentaje de error en test.

Implementación de las metaheurísticas

En este apartado se va a explicar como están implementadas las *metaheurísticas* que utilizamos en el *Trabajo de Fin de Grado*. Se han utilizado cuatro entre todas las que tiene *Pygmo2*, todas ellas bioinspiradas:

- Algoritmo Genético.
- Particle Swarm Optimization.
- Ant Bee Colony.
- Simulated Annealing.

A nivel de implementación, todas funcionan de la misma manera y siguen un proceso idéntico en *Pygmo2* variando únicamente los *hiper-parámetros* con los que se definen. Por ello, describiré el proceso que se ha seguido para el *Algoritmo Genético* que puede ser reproducido únicamente cambiando la función de la *metaheurística*.

En primer lugar es necesario definir inicializar el problema a optimizar, en este caso se va a inicial la función de *Booth*, que muestra la figura A.6. El siguiente paso es la definición de la *metaheurística* que se va a utilizar, definida como parámetro de la función *pg.algorithm()*. Para definir un algoritmo genético se utiliza la función *pg.sga()*, y como parámetros más relevantes serían los siguientes:

- Gen: Número de generaciones.
- Cr: Probabilidad de cruce del *Algoritmo Genético*.
- M: Probabilidad de mutación.

Esos tres parámetros son los más relevantes a la hora de definir un algoritmo genético. Lo siguiente es definir la población inicial, que irá evolucionando durante el número de generaciones definido como *hiperparámetro*, para realizar este proceso será necesario pasarle como parámetro a la función *pg.population()*, el problema y el número de individuos, que representa el tamaño de la población. El último paso sería llamar a *algorithm.evolve()* que recibe la población inicial y evoluciona esa población durante el número de generaciones previamente definido. Para obtener el mejor individuo se utiliza *population.champion_f* y para los valores de las variables dependientes o *hiperparámetros*, en función del problema, se pueden obtener con *population.champion_x* que devuelve un vector con esos valores.

5.3.2. SMAC3

En este apartado se va a mostrar y a explicar la utilidad de la librería SMAC3 [3]. En este proyecto se ha utilizado para el proceso de Optimización Bayesiana sobre los distintos *hiper-parámetros* de las *metaheurísticas*, la parte llamada *Optimización Bayesiana* de la figura 4.2.

Esta herramienta permite la configuración de algoritmos y para ello utiliza *Optimización Bayesiana* y un algoritmo *Random Forest*. A lo largo de esta sección se mostrará la implementación que hemos realizado nosotros para resolver la búsqueda de doble nivel relacionando las ideas de la sección 5.3.1 con las de esta y mostrando como optimizar doblemente funciones matemáticas y *Algoritmos de Aprendizaje Automático*. Se mostrarán los distintos fragmentos de código utilizados para la doble optimización de la función de *Booth*, el funcionamiento es análogo para el resto de algoritmos y funciones.

Antes de comenzar a comentar la implementación realizada es necesario realizar una descripción general de como funciona una implementación de este estilo en *SMAC3*. En primer lugar es necesario instanciar el espacio de configuración, *ConfigurationSpace()* en *SMAC3*, este objeto almacenará todos los parámetros que queremos optimizar y sus dominios. Lo siguiente por tanto, será definir esos parámetros y las distintas restricciones que pueda haber entre ellos. Es necesario definir también una función para evaluar los parámetros definidos y que devuelva un valor a modo de puntuación. Se instancia el objeto *Scenario*, que define el número máximo de veces que se va a ejecutar la función de evaluación y el tipo de optimización que se va a realizar, de forma general, configura *SMAC3*. Por último, se inicializa el objeto *SMAC4HPO*, con el escenario, la función de evaluación y la semilla como entradas, para inciciar la optimización es necesario llamar al método *optimize()*, que devolverá un objeto con la configuración del valor optimizado.

En la figura A.1 se muestra la definición de las metaheurísticas, sus parámetros, restricciones y su adicción al *ConfigurationSpace*. Las restricciones son necesarias ya que las *metaheurísticas* tienen distintos parámetros y hay que poder seleccionar unos u otros en función de la que se este ejecutando. Se definen también los valores máximo y mínimo que pueden tomar, si son variables continuas y las distintas posibilidades que tiene si son categóricos. De esta manera se incorpora el primer nivel de optimización al segundo. La única metaheurística que no tiene ningún parámetro es *Simulated Annealing*, que se ejecuta con los que tiene por defecto. Esta parte es común tanto para funciones matemáticas como para algoritmos.

La figura A.2 representa la inicialización del *Scenario*, que como se ha comentado anteriormente configura la optimización. En ella se define el parámetro *quality* que indica que se optimiza el valor y no el tiempo de ejecución, *runcount-limnit* es el número máximo de evaluaciones, que sea determinista y por último también necesita el *ConfigurationSpace*. El siguiente paso es inicializar el objeto *SMAC4HPO* y llamar a la función *optimize()* que devolverá el *incumbent* conteniendo un conjunto de estructuras del fin de la evaluación.

Por último, es necesario guardar el resultado que nos interese, en el caso de las funciones, el valor optimizado y la configuración de la *metaheurística* con la que se obtiene. En el caso de los *Algoritmos de Aprendizaje automático* serán guardados otros parámetros además de su valor, pero se explicará un poco más abajo.

El último elemento es la función de evaluación, su implementación queda reflejada en la figura

A.3. Esta función es la más importante del código y la que conecta *SMAC* y *Pygmo2* y aglutina la funcionalidad de la búsqueda de doble nivel que tantas veces se ha mencionado. El proceso es análogo al explicado en la sección 5.3.1. Esta figura representa la optimización de la función de *Booth* y únicamente es necesario que devuelva el valor que resulta de la optimización metaheurística.

En el caso de los algoritmos de *aprendizaje automático*, lo único que cambia son los datos que se devuelven como muestra la figura A.4. El objetivo final del trabajo es conseguir los *hiperparámetros* optimizados, para ello, necesitamos que cuando el algoritmo termine se devuelvan los valores de los *hiperparámetros* que se han utilizado en la obtención del valor optimizado. La figura A.4 muestra el caso de *XGBoost* pero es análogo para la *Red Neuronal Convolucional* y el *Perceptrón Multicapa*.

Para almacenar está información en los algoritmos se utiliza el mismo método que en la figura A.2, pero añadiendo al fichero que guarda el resultado esos *hiperparámetros*. El proceso puede observarse en la figura A.5.

5.4. Algoritmos de Aprendizaje Automático

Este es el último apartado de la implementación. En él se va a explicar de forma general como funcionan los algoritmos de *Aprendizaje Automático* que se han utilizado para la optimización de *hiperparámetros*. Las implementaciones son muy sencillas, ya que como se ha comentado en otras secciones, no se contaba con la capacidad de cómputo suficiente para realizar experimentos reales. Por tanto, suponen una aproximación a una aplicación real del desarrollo, para demostrar que es totalmente funcional y permite la optimización tal y como ha sido planteada.

Por último, se va a explicar la implementación de la *Red Neuronal Convolucional* aunque finalmente no se han podido ejecutar los experimentos diseñados sobre ella por falta de potencia de computo, si se ha comprobado su funcionamiento en pruebas de implementación muy reducidas y ha supuesto una gran cantidad de tiempo de trabajo.

5.4.1. XGBoost

El algoritmo *XGBoost* [14] es una librería de *Gradient Boosting* optimizado, que está diseñada para ser eficiente, flexible y portable. Además tiene gran capacidad de funcionamiento distribuido que la hace una gran opción para resolver problemas de *Data Science* de forma óptima y rápida actualmente.

El *Gradient Boosting*, es una técnica de *Aprendizaje Automático* que se basa en construir un modelo de predicción robusto formado por un conjunto de modelos de predicción débiles, como pueden ser *árboles de decisión*. Otro ejemplo de un algoritmo que funcione de esta manera sería *Random Forest*.

Se puede ver la implementación de este algoritmo en la figura A.7, se explicará su funcionamiento

en base a lo que aparece en ella.

En primer lugar se generan las particiones de *train* y *test* y se convierten a un tipo especial de datos *DMatrix* propio de la libreria *XGBoost*. Posteriormente es necesario inicializar los hiper-parámetros que utiliza el modelo, se usa el *softmax* como optimizador objetivo, el número de clases es 10 y depende del fichero de datos. Los demás parámetros son los que se van optimizar, la constante de aprendizaje *eta*, que está en el rango 0 a 1, la profundidad máxima hasta la que pueden expandirse los árboles *max_depth* que hemos definido un mínimo de 0 y un máximo de 6, para encontrar la óptima y que el problema no se expandiera del todo para evitar sobrepasar la capacidad del ordenador y el número de rondas de ejecución *num_round*, cuyo suelo es 0 y el techo 20.

Una vez definidos los parámetros es necesario definir una lista de observación *watchlist*, cuya función es la de almacenar información durante la ejecución de las variables *xg_test* y *xg_train*. No tienen ninguna utilidad para la implementación pero es necesario definirla.

El siguiente paso es definir el modelo, utilizando para ello la función *xgb.train()* cuyos argumentos son el diccionario *params*, la matriz con los datos de entrenamiento, el número de rondas, la lista de observación *watchlist* y la verbosidad, que se define a *False* para aumentar la velocidad de ejecución.

Por último, calculamos la predicción del modelo entrenado utilizando *xgb.predict()* sobre la matriz de datos de test *xgtest*, que se compara con los datos reales *y_test* devolviendo el porcentaje de error.

5.4.2. Perceptrón Multicapa

La segunda aplicación real se ha implementado sobre un *Perceptrón Multicapa* con una capa oculta. Es necesario saber que es una red neuronal artificial formada por múltiples capas, una de entrada, una de salida y un número definido de capas ocultas. Esta red cuenta con dos fases, la *propagación*, que permite el cálculo de la salida de la red mandando los valores desde la capa de entrada hasta la de salida, pasando por las intermedias y la *backpropagation*, en la que los errores obtenidos en la capa de salida se propagan hacia la de entrada con el objetivo de ajustar los pesos de las conexiones para que el valor que estime la red sea lo más próximo al real posible.

El aprendizaje de esta red está basado en la minimización de la función gradiente del error que se produce entre las salidas predichas y las reales. Otra de las características de la red es que necesita tener funciones de transferencia derivables.

Se ha decidido utilizar esta red porque es un aproximador universal, pudiendo resolver cualquier problema, aunque no sea separable de forma lineal. Una de sus limitaciones es la aparición de mínimos locales en la función de error, ya que cuando se detecta un mínimo en la función de error el entrenamiento finaliza, obviando el mínimo global y no alcanzando el punto de convergencia. A pesar

de esto y aunque existen redes más potentes hoy en día, es una buena elección como aproximación a problema real para realizar experimentos.

Los primeros intentos fueron su implementación en *Keras* [15], un framework de *Deep Learning* con un uso popularizado en los últimos años pero no fue posible, ya que está librería está diseñada para ejecutarse de forma óptima en *GPU* y no se disponían de los medios necesarios. Además una sesión del entorno *Google Colab* no era suficiente. Por lo expuesto anteriormente, fue necesario buscar alternativas.

La mejor alternativa fue hacer uso de Scikit-Learn [16] y su implementación del *Perceptrón Multicapa*. Esta implementación se muestra en figura A.10.

Se define el conjunto de datos y antes de dividir en particiones se hace un preprocesado de los datos, utilizando *StandardScaler*, que normaliza los datos utilizando su media y desviación típica. Este proceso ayuda a la convergencia del modelo. Lo siguiente es la subdivisión en *train* y *test*.

El modelo de *Perceptrón Multicapa* utilizado es *MLPClassifier* y se ha definido con los parámetros por defecto, exceptuando el tamaño de la capa oculta que va de 6 a 64 neuronas, la constante de aprendizaje, que toma valores entre 0.00001 y 0.1. El último *hiperparámetro* que se optimiza es el número de épocas con valor mínimo 250 y máximo 10000.

Utilizando el método *mlp.fit()* junto con la partición de *train* obtenemos un modelo listo para realizar predicciones en *test* con la función *mlp.predict()*.

Por último, devolverá el porcentaje de error utilizando la función *accuracy_score()*, que compara las predicciones con las clases reales.

5.4.3. Red Neuronal Convolucional

El último ejemplo de algoritmo que vamos a presentar es una *Red Neuronal Convolucional*, como se ha mencionado no se han podido realizar las pruebas por falta de capacidad de cómputo pero, es una implementación correcta y de mucho interés como trabajo futuro. Este apartado será más breve que los anteriores y se realizará una explicación mínima.

Este tipo de red pertenece al aprendizaje supervisado e intenta imitar al cortex visual del ojo humano, identificando características que le permitan identificar objetos, de forma más general permite implementar visión artificial. Contienen varios tipos de capas que detectan lineas y curvas, que se van especializando hasta llegar a capas más profundas que permiten reconocer formas más complejas.

Las capas convencionales son un filtro sobre la matriz de píxeles que representa la imagen, generando una nueva imagen pero filtrada, obteniendo sus características más representativas.

Para realizar un reconocimiento de imágenes utilizando esta técnica es necesario utilizar otro tipo

de capas además, de las explicadas, pero en este caso con comprender que es una convolución resulta suficiente.

Para su implementación utilizamos *Keras* [15], aunque hay otras alternativas esta es la más sencilla. La implementación funcionará sobre el dataset de imágenes *MNIST*, muy conocido.

En la figura A.9 se muestra la implementación, en este caso un poco distinta con respecto a *XGBoost* y el *Perceptrón Multicapa*, ya que, se define un constructor para el problema. La explicación a esta modificación es que en el caso de los demás se carga el *dataset* en cada iteración, al no ser muy pesado, pero el *MNIST* si lo es, entonces para evitarlo, se carga una vez y a la hora de construir el objeto se inicializan los subconjuntos de *train* y *test* directamente.

El siguiente paso es construir el modelo, que cuenta con dos capas convoluciones *Conv2D*, una capa *Flatten*, que aplana las matrices, convirtiéndolas en una lista y por último una capa densa que representa la capa de salida. Las funciones de activación de las convoluciones son de tipo *relu*, que devuelve el mayor máximo entre 0 y la entrada que reciba, siendo una función lineal. La última capa, sin embargo utiliza *softmax*, más conveniente para predecir una clase final de salida.

Los hiper-parámetros optimizados son el número de capas de las capas convolucionales, el tamaño de los *kernels* que se le aplican a las funciones y por último el número de épocas de entrenamiento, todos ellos entre valores 1 y 5. El número de neuronas de las capas convolucionales va multiplicado por un factor 16 en el caso de la primera y 8 en la segunda.

Los demás pasos son los habituales, inicialización del modelo utilizando *compile()*, entrenamiento *fit()* con el subconjunto correspondiente y el mismo caso para el test, con la función *evaluate()*. Por último se devolverá el porcenaje de error de las predicciones generadas.

6

Experimentos

En este capítulo se van a mostrar y explicar los distintos experimentos realizados para validar el correcto funcionamiento de la aplicación diseñada, se realizará una breve descripción de los experimentos de *XGboost*, el *perceptrón multicapa* y por último, una de las funciones matemáticas implementadas. Posteriormente se presentarán gráficas y tablas con los resultados y una breve conclusión final sobre ellos.

6.1. Descripción

En esta sección se van a describir todos los experimentos realizados para obtener los resultados que se exponen en la siguiente sección del documento. En primer lugar, es necesario volver a reseñar las limitaciones temporales y de capacidad de cómputo lo que ha provocado que los experimentos únicamente una aproximación a lo que se podría conseguir con la herramienta propuesta.

Cada ejecución del programa sobre un algoritmo o una función matemática devolverá el mejor valor obtenido a lo largo de la ejecución, junto con la metaheurística e hiperparámetros que lo obtienen. En este proceso la *optimización bayesiana* va realizando iteraciones, probando con distintas configuraciones de *metaheurísticas* hasta converger o alcanzar las iteraciones definidas inicialmente en la configuración del programa, para el problema definido de forma inicial.

En el caso de estudio, los algoritmos *XGBoost* y el *perceptrón multicapa* realizan 25 iteraciones de *optimización bayesiana* de los que se obtienen 25 valores y configuraciones de *metaheurísticas*. Esto comprendería una ejecución pero, para poder obtener resultados concluyentes, es necesario ejecutar varias veces el programa y que además, sea posible reproducirlos. Para ello, se utiliza la semilla aleatoria, que permite que para una misma semilla los resultados del programa sean los mismos en distintas ejecuciones. Esta semilla, solo se indica en la *optimización bayesiana*, ya que a la hora de definir los algoritmos se utilizan siempre las mismas particiones, tanto para entrenar como para validar, si esto no fuera así, sería necesario definirla también.

Una vez descrito el proceso que se utiliza para generar las ejecuciones es necesario explicar como se obtienen los datos presentados en las gráficas del siguiente apartado. Cada ejecución del pro-

grama reportará 25 valores, que se corresponden con el mejor valor observado en cada iteración de *optimización bayesiana*. El resultado en cada iteración será el mejor valor observado hasta ese momento, contemplando también las iteraciones anteriores, esto se ve de forma más sencilla utilizando un ejemplo. Supongamos un programa con 4 iteraciones de *optimización bayesiana*, que dan 8,4,6,2 como resultado final, esto se correspondería con los valores aislados obtenidos en cada iteración, pero queremos que esta lista contenga siempre el mejor valor observado hasta el momento, por tanto, la salida sería 8,4,4,2. El resultado final es el mejor valor observado en cada semilla. Por último, hay que promediar esos valores para cada una de las semillas y obtener la desviación típica, lo que nos devuelve finalmente la media de los valores para cada semilla y cada iteración, generando una gráfica decreciente.

Además de presentar las gráficas, mostramos una tabla con la configuración de la heurística que mejor valor obtiene, para cada una de las semillas. Esto es bastante útil, ya que uno de los objetivos del trabajo es ver que *metaheurística* nos devuelve la configuración óptima del algoritmo y con esta tabla puede observarse de manera rápida así como los *hiperparámetros* con los que se obtiene. De igual forma, el programa genera también un fichero de salida con los valores de configuración del algoritmo con los que se obtiene el mejor resultado, aunque estos datos no se incluyen en las tablas, ya que se obtienen directamente al aplicar la metaheurística optimizada de la que se ha hablado anteriormente.

La última parte de los experimentos es realizar un número de búsquedas aleatorias sobre el mismo espacio sobre el que se realiza la doble optimización y comparar los resultados, los resultados obtenidos se presentan en la misma tabla que los resultados, para poder ver la comparación.

La gráfica y tabla únicamente se presentan para los experimentos de *XGBoost*, en el *perceptrón multicapa* se muestra el mejor valor obtenido por la optimización comparado con las búsquedas aleatorias, además de extraer conclusiones en base a ellos.

En el caso de la optimización de funciones, que se propuso como experimento inicial, se minimiza el valor de la función, intenta encontrar su mínimo global. En los resultados se muestra una gráfica generada de la misma manera que para los algoritmo de *aprendizaje automático* propuestos. No se profundiza demasiado en ellas a lo largo del trabajo ya que han sido una herramienta que ha permitido llegar al resultado final. El número de iteraciones de *optimización bayesiana* es de 100 en este caso, ya que estos problemas son mucho más rápidos que los algoritmos, lo que se acercaría más a un caso real.

6.2. Resultado de los experimentos

A continuación se muestran los resultados obtenidos para el *perceptrón multicapa*, el algoritmo *XGboost* y la función matemática *Branin*. Esta sección se subdivide en tres apartados, dedicados a cada una de ellas, se presentarán las gráficas, tablas y finalmente se extraerá una conclusión en base

a esos resultados presentados.

Otro aspecto importante a comentar es que se ha utilizado para los algoritmos de *aprendizaje automático* el dataset *Digits* que es un pequeño dataset con números del 0 al 9 manuscritos y cuenta con un total de 1800 datos. En general un dataset muy pequeño pero muy útil para las pruebas de concepto que han sido realizadas.

6.2.1. XGBoost

La gráfica 6.1 y la tabla 6.1 muestran los resultados siguiendo el procedimiento explicado en el apartado 6.1.

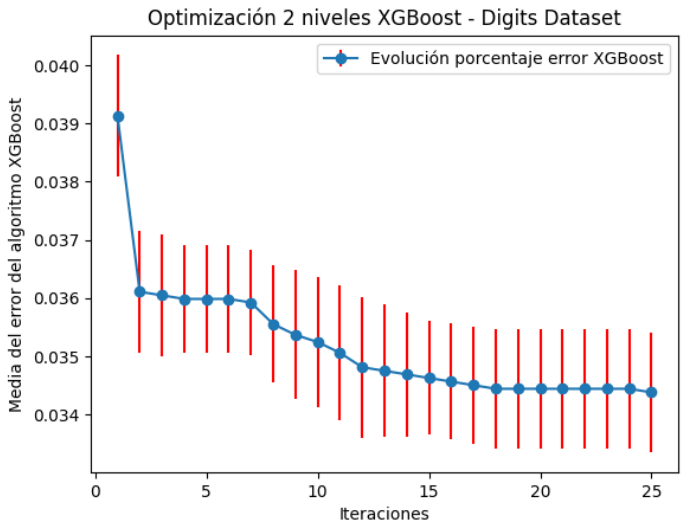

Figura 6.1: Gráfica que muestra la evolución del mínimo error obtenido en cada iteración de *optimización bayesiana* aplicado al *XGBoost*.

Podemos observar en la gráfica 6.1 que el valor en este caso sigue la lógica que indica un proceso en el que se utiliza *optimización bayesiana* para la optimización de un porcentaje de error, el error va descendiendo hasta que sobre la iteración 18 empieza a estabilizarse o a descender mínimamente, lo que indica que ha convergido o el punto está muy cercano. Lo ideal habría sido darle de un número mayor de iteraciones para que hubiera podido converger, pero no ha sido posible por el coste computacional.

En la tabla 6.1 se pueden ver, de nuevo, los mejores valores obtenidos, junto a su *metaheurística* e *hiperparámetros*. Si comparamos los valores obtenidos con las búsquedas aleatorias, existe una mínima diferencia aunque no es relevante como para extraer conclusiones, ni positivas ni negativas.

Semilla aleatoria	Metaheurística	Num. generaciones	Parámetro 1	Parámetro 2	Parámetro 3	Error	Error búsq. aleat.
1	Particle swarm	10	max vel = 0.982	neighb param = 84	omega = 0.817	**0.035**	0.041
2	Particle swarm	8	max vel = 0.718	neighb param = 6	omega = 0.37	**0.033**	0.043
3	Particle swarm	10	max vel = 0.789	neighb param = 94	omega = 0.414	**0.033**	0.038
4	Particle swarm	10	max vel = 0.453	neighb param = 98	omega = 0.852	**0.033**	0.041
5	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.039
6	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.041
7	Genetic Algorithm	4	cr = 0.874	m = 0.13	selection = truncated	**0.033**	0.041
8	Particle swarm	5	max vel = 0.771	neighb param = 25	omega = 0.935	**0.033**	0.038
9	Genetic Algorithm	8	cr = 0.673	m = 0.058	selection = tournament	**0.035**	0.038
10	Particle swarm	3	max vel = 0.992	neighb param = 87	omega = 0.492	**0.035**	0.039
11	Particle swarm	9	max vel = 0.506	neighb param = 93	omega = 0.695	**0.033**	0.038
12	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.039
13	Particle swarm	10	max vel = 0.463	neighb param = 13	omega = 0.774	**0.033**	0.039
14	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.041
15	Ant Bee Colony	6	acc = 0.505	ker = 55	q = 2.62	**0.033**	0.037
16	Particle swarm	8	max vel = 0.982	neighb param = 33	omega = 0.559	**0.033**	0.037
17	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.039
18	Simulated Annealing	4	Default	Default	Default	**0.035**	0.039
19	Particle swarm	10	max vel = 0.966	neighb param = 23	omega = 0.552	**0.033**	0.039
20	Particle swarm	8	max vel = 0.723	neighb param = 40	omega = 0.426	**0.033**	0.039
21	Particle swarm	9	max vel = 0.502	neighb param = 96	omega = 0.362	**0.033**	0.039
22	Particle swarm	6	max vel = 0.775	neighb param = 35	omega = 0.553	**0.033**	0.041
23	Ant Bee Colony	10	acc = 0.078	ker = 60	q = 3.21	**0.035**	0.039
24	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.039
25	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.037
26	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.035**	0.037
27	Ant Bee Colony	8	acc = 0.268	ker = 46	q = 2.78	**0.035**	0.039
28	Particle swarm	8	max vel = 0.567	neighb param = 63	omega = 0.264	**0.033**	0.035
29	Particle swarm	8	max vel = 0.775	neighb param = 26	omega = 0.325	**0.033**	0.039
30	Particle swarm	6	max vel = 0.316	neighb param = 49	omega = 0.895	**0.035**	0.041

Tabla 6.1: Mejores metaheurísticas con sus hiperparámetros para cada una de las semillas aleatorias, para *XGboost*.

6.2.2. Perceptrón multicapa

En este caso se ha realizado el mismo tipo de experimento que para *XGBoost*.

Las medias del mínimo valor observado, a lo largo de las iteraciones de la *optimización bayesiana*, se han mantenido constantes, lo que quiere decir que en la primera iteración se alcanza el mejor valor posible en la primera iteración para todos los casos, además, la desviación típica para todos ellos es 0, lo que quiere decir que el mejor valor se mantiene constante en todos los casos.

El mejor valor obtenido para cada una de las 30 semillas es de 0.022 y los valores aleatorios obtenidos se encuentran entre 0.024 y 0.030, de nuevo, una diferencia que no permite extraer conclusiones relevantes pero que indica que el resultado de la *optimización bayesiana* es ligeramente mejor.

Los resultados del experimento no han sido sido satisfactorios, pero para ejemplificar que una *red neuronal* también podría ser optimizada es suficiente. Los resultados si se hubiera podido optimizar una *red neuronal convolucional* deberían haber sido mejores y más representativos pero dada la potencia necesaria no ha sido posible llevarlos a cabo.

6.2.3. Función *branin*

Este experimento es uno de los que sirvió de base para el resto del trabajo como se ha comentado anteriormente, no es el objetivo final, pero se ha considerado relevante mostrarlo. En la gráfica 6.2 podemos ver la evolución de la optimización del mínimo de la función utilizando *optimización bayesiana*, a lo largo de 100 iteraciones y *metaheurísticas* bioinspiradas, de esas 100 iteraciones únicamente se muestran las primeras 50, ya que el resto son iguales y permite visualizar mejor la gráfica.

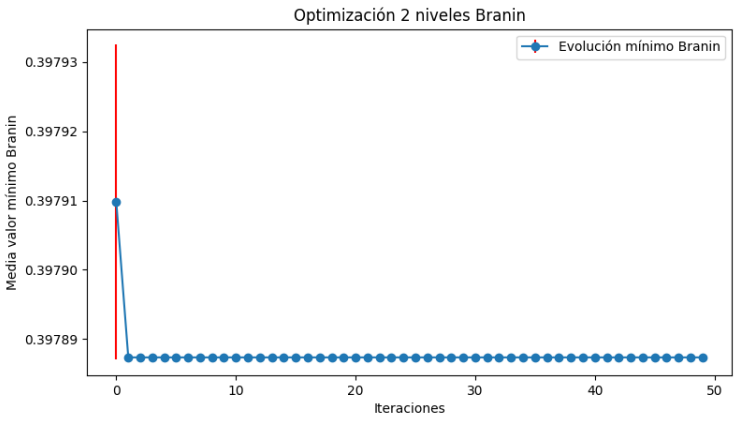

Figura 6.2: Gráfica que muestra la evolución del valor minimizado de la función *Branin* obtenido en cada iteración de *optimización bayesiana*.

Como se puede observar en la figura 6.2, la evolución es muy parecida a la obtenida con el *perceptrón*, con una diferencia, que necesita una iteración para alcanzar el mínimo de la función. En la primera iteración vemos como el valor se sitúa un poco por encima del mínimo y encontramos también líneas representando la desviación típica. Esto quiere indicar que en esa iteración las 100 distintas semillas obtienen valores mínimos diferentes y a partir de ese momento, todas las semillas encuentran el mínimo en la iteración 2.

El valor mínimo de la función *Branin* es 0.397887 que es el que alcanza la optimización realizada, tal y como puede observarse en la gráfica 6.2.

Este ejemplo no es relevante para sacar conclusiones pero se puede ver que el programa implementado tiene la capacidad de minimizar funciones matemáticas, esta es muy sencilla, pero podría hacerlo en casos de mayor complicación.

7

Conclusiones y trabajo futuro

En este último apartado de la memoria se va a realizar una reflexión sobre todo el trabajo realizado, extraer conclusiones en base a los experimentos y comentar el posible trabajo futuro que conllevaría este proyecto.

En primer lugar, destacar el gran aprendizaje que ha supuesto el enfrentamiento a una técnica de optimización desconocida previamente, la *optimización bayesiana*. El trabajo también ha permitido una mayor comprensión de lo que es una *metaheurística*, más concretamente de aquellas bioinspiradas y darse cuenta de las utilidades que pueden tener en la aplicación sobre problemas del mundo real.

La herramienta planteada genera una doble optimización, como ya se ha comentado en otras secciones, lo que conllevó un gran trabajo para conseguir encajar las piezas y que funcionaran correctamente de manera conjunta. Además, las librerías de *python* utilizadas, *SMAC3* y *pygmo2* no tienen mucho uso y es complicado encontrar documentación en internet que no sea la propia de los desarrolladores, en ese sentido, se ha necesitado bastante tiempo para comprender su funcionamiento y conseguir desarrollar la herramienta de forma que funcionará correctamente.

Inicialmente se planteaban 3 objetivos, que pueden verse en la sección 3. El primero de ellos (**o.1**) era la obtención de una herramienta que permitiera la optimización de *hiperparámetros* de algoritmos de *aprendizaje automático* por medio de una búsqueda de dos niveles, el primero basado en *metaheurísticas* bioninspiradas y el segundo *optimización bayesiana* sobre los resultados previamente optimizados por el nivel 1. El 3er objetivo (**o.3**) tenía que ver con la obtención de una recomendación de *metaheurística*, junto con los valores de sus *hiperparámetros* que permitiera optimizar más exhaustivamente el problema planteado, en un futuro.

Estos dos objetivos se han cumplido de forma correcta, el programa generado optimiza en dos niveles, codificando previamente el problema deseado y finalmente devuelve un fichero en el que se imprime la *metaheurística* e *hiperparámetros* que obtienen el mejor valor a lo largo de la ejecución.

El segundo objetivo (**o.2**) y último, era realizar una comparación entre los resultados obtenidos por el programa y una búsqueda aleatoria en el mismo espacio de búsqueda. Este, en cambio, no ha sido satisfecho totalmente. En el caso del *perceptrón multicapa*, tal y como se muestra en la tabla **??**, no

hay diferencias entre el valor optimizado y el aleatorio. En el caso de *XGBoost* ocurre algo similar, el error optimizado es menor que el error de las búsquedas aleatorias pero una cantidad ínfima.

Esto provoca que no se pueda llegar a la conclusión de que la búsqueda doble implementada mejore a una búsqueda aleatoria. Este comportamiento ha podido ocurrir por varias razones, la primera es la simplicidad de los problemas. Se ha utilizado el dataset de *digits*, muy útil en un entorno educativo, pero que no representa un problema real. Por tanto, al ser un problema sencillo, se considera normal que los resultados de una búsqueda aleatoria sean similares a los del valor optimizado, ya que el espacio de búsqueda no es lo suficientemente complejo y cualquier algoritmo sin gran configuración podría llegar a unos resultados similares.

En el *perceptrón* y *XGBoost*, las *metaheurísticas* optimizan 3 valores de *hiperparámetros*, de todos los que tienen estos algoritmos, es posible también que estos *hiperparámetros*, elegidos sin un estudio previo del problema, no sean los que más afectan al dataset *digits* y quizá optimizando otros se obtendrían mejores resultados.

Como última posibilidad también se contempla un posible error del código en algún punto, aunque se considera improbable, ya que el funcionamiento del programa ha sido comprobado varias veces y funciona según lo establecido.

En resumen, los problemas elegidos no son lo suficientemente complicados como para obtener diferencias y los *hiperparámetros* elegidos quizá tampoco sean los más relevantes en ellos. En estos puntos se centra el trabajo futuro que podría plantearse, en primer lugar, si se tiene la capacidad de cómputo adecuada, sería conveniente tratar de probar la herramienta sobre un dataset más grande, como podría ser el *Mnist*. Además las *redes neuronales convolucionales* se encuentran hoy en día en un gran auge y sería deseable conseguir obtener resultados en base a la implementación realizada que no ha sido posible ejecutar.

Por último, sería deseable implementar algún nuevo problema actual, como ejemplo podría ser interesante el uso de la herramienta sobre una red neuronal de tipo *GAN*, que actualmente está siendo utilizada por la empresa *Nvidia* en multitud de proyectos relacionados con visión artificial y videojuegos.

Bibliografía

[1] L. Cornejo-Bueno, E. C. Garrido-Merchán, D. Hernández-Lobato, and S. Salcedo-Sanz, "Bayesian optimization of a hybrid system for robust ocean wave features prediction," *Neurocomputing*, vol. 275, pp. 818–828, 2018.

[2] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[3] M. Lindauer, K. Eggensperger, M. Feurer, S. Falkner, A. Biedenkapp, and F. Hutter, "Smac v3: Algorithm configuration in python." `https://github.com/automl/SMAC3`, 2017.

[4] F. Biscani, D. Izzo, and M. Märtens, "esa/pagmo2: pagmo 2.7," Apr. 2018.

[5] X.-S. Yang, "A new metaheuristic bat-inspired algorithm," in *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pp. 65–74, Springer, 2010.

[6] A. Rajagopalan, D. R. Modale, and R. Senthilkumar, "Optimal scheduling of tasks in cloud computing using hybrid firefly-genetic algorithm," in *Advances in Decision Sciences, Image Processing, Security and Computer Vision*, pp. 678–687, Springer, 2020.

[7] E. C. G. Merchán and L. C. J. Pérez, "Towards automatic bayesian optimization: A first step involving acquisition functions," *arXiv preprint arXiv:2003.09643*, 2020.

[8] C. Huang, B. Yuan, Y. Li, and X. Yao, "Automatic parameter tuning using bayesian optimization method," in *2019 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2090–2097, IEEE, 2019.

[9] F. He, J. Zhou, Z.-k. Feng, G. Liu, and Y. Yang, "A hybrid short-term load forecasting model based on variational mode decomposition and long short-term memory networks considering relevant factors with bayesian optimization algorithm," *Applied energy*, vol. 237, pp. 103–116, 2019.

[10] S. Sano, T. Kadowaki, K. Tsuda, and S. Kimura, "Application of bayesian optimization for pharmaceutical product development," *Journal of Pharmaceutical Innovation*, pp. 1–11, 2019.

[11] H.-P. Nguyen, J. Liu, and E. Zio, "A long-term prediction approach based on long short-term memory neural networks with automatic parameter optimization by tree-structured parzen estimator and applied to time-series data of npp steam generators," *Applied Soft Computing*, vol. 89, p. 106116, 2020.

[12] M. Zhao and J. Li, "Tuning the hyper-parameters of cma-es with tree-structured parzen estimators," in *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*, pp. 613–618, IEEE, 2018.

[13] T. Hathaway, *Data Flow Diagrams - Simply Put!: Process Modeling Techniques for Requirements Elicitation and Workflow Analysis*. CreateSpace Independent Publishing Platform, 2016.

[14] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016.

[15] F. Chollet *et al.*, “Keras.” https://keras.io, 2015.

[16] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, “API design for machine learning software: experiences from the scikit-learn project,” in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

Apéndices

A

CÓDIGO IMPEMENTADO

En este apéndice se van a mostrar los fragmentos de código referenciados en el capítulo 5.

```
# Building Config. Space, whice defines all parameters and their ranges.
cs = ConfigurationSpace()

# Metaheuristcs to use in the bayesian optimization, defined has a string
metaheuristics = CategoricalHyperparameter("metaheuristic", ["genetic_algorithm",
                           "particle_swarm_opt", "ant_bee_colony", "simulated_annealing"],
                           default_value = "genetic_algorithm")

# Number of generations defined globally, for the three metaheuristics
gen = UniformIntegerHyperparameter("gen", 20, 1000, default_value = 20)

# Definition of parameters for each metaheuristic.
# Genetic Algorithm Parameters
cr = UniformFloatHyperparameter("cr", 0.6, 0.9, default_value = 0.90)
m = UniformFloatHyperparameter("m", 0.02, 0.15, default_value = 0.02)
selection = CategoricalHyperparameter("selection", ["tournament", "truncated"], default_value = "tournament")

# Particle Swarm Parameters
omega = UniformFloatHyperparameter("omega", 0.1, 1.0, default_value = 0.7298)
neighb_param = UniformIntegerHyperparameter("neighb_param", 1,100, default_value = 4)
max_vel = UniformFloatHyperparameter("max_vel", 0.1, 1.0, default_value = 0.5)


# Ant colony Parameters
ker = UniformIntegerHyperparameter("ker", 10, 100, default_value = 63)
acc = UniformFloatHyperparameter("acc", 0.01, 1.0, default_value = 0.01)
q = UniformFloatHyperparameter("q", 0.1, 5.0, default_value = 1.0)

# Adding this parameters to Config. Space
cs.add_hyperparameters([gen, metaheuristics, cr, m, selection, omega, neighb_param, max_vel,
                        ker, acc, q])

# Setting constraints, each parameter should activate when his model is chosen
# It's neccesary to a condition for each parameter, so we need 9.
use_cr = InCondition(child = cr, parent = metaheuristics, values=["genetic_algorithm"])
use_m = InCondition(child = m, parent = metaheuristics, values=["genetic_algorithm"])
use_selection = InCondition(child = selection, parent = metaheuristics, values=["genetic_algorithm"])

use_omega = InCondition(child = omega, parent = metaheuristics, values = ["particle_swarm_opt"])
use_neighb_param = InCondition(child = neighb_param, parent = metaheuristics, values = ["particle_swarm_opt"])
use_max_vel = InCondition(child = max_vel, parent = metaheuristics, values = ["particle_swarm_opt"])

use_ker = InCondition(child = ker, parent = metaheuristics, values = ["ant_bee_colony"])
use_acc = InCondition(child = acc, parent = metaheuristics, values = ["ant_bee_colony"])
use_q = InCondition(child = q, parent = metaheuristics, values = ["ant_bee_colony"])
# We add the conditions
cs.add_conditions([use_cr, use_m, use_selection, use_max_vel, use_omega, use_neighb_param,
                use_q, use_ker, use_acc])
```

Figura A.1: Definición de *metaheurísticas*, hiper-parámetros y restricciones en *SMAC3*.

```python
# Scenario object
scenario = Scenario({"run_obj": "quality",   # we optimize quality (alternatively runtime)
                     "runcount-limit": int(sys.argv[2]),   # max. number of function evaluations; for this example set to a low number
                     "cs": cs,               # configuration space
                     "deterministic": "true"
                     })

# Building SMAC object to start optimization
print("Beginning optimization...")
smac = SMAC4HPO(scenario = scenario, rng = np.random.RandomState(int(sys.argv[1])),
                tae_runner = evaluation_function)

incumbent = smac.optimize()
inc_value = evaluation_function(incumbent)

print("Optimized value: %.2f" % (inc_value))

# Writing the final result to a file, to read it later
result_file_name = "seed_" +sys.argv[1] +"_" +sys.argv[2] +"_iterations_value.txt"
final_result = "Optimized value" +str(inc_value)
with open(result_file_name, "w") as f:
    f.write(str(incumbent))
    f.write(final_result)
```

Figura A.2: Inicialización de objetos, número de iteraciones, función de evaluación y guardado de objetos de la función de *Booth* en *SMAC3*.

```python
def evaluation_function(cfg):
    """
        Evaluation function
    """

    # Definition of the problem
    prob = pg.problem(booth_function())

    cfg = {k : cfg[k] for k in cfg if cfg[k]}

    # Elimination of the metaheuristic, dont needed to initiate algorithms
    metaheuristic = cfg.pop("metaheuristic", None)

    # Configutation of the evaluation function to take care depending on the iteration of the alg.
    if metaheuristic == 'genetic_algorithm':
        algo = pg.algorithm(pg.sga(**cfg))
        population = pg.population(prob,int(sys.argv[3]))
        population = algo.evolve(population)

    elif metaheuristic == 'ant_bee_colony':
        algo = pg.algorithm(pg.gaco(**cfg))
        population = pg.population(prob,int(sys.argv[3]))
        population = algo.evolve(population)

    elif metaheuristic == 'simulated_annealing':
        algo = pg.algorithm(pg.simulated_annealing(Ts = 10., Tf = .1, n_T_adj = 10, n_range_adj = 10, bin_size = 10, start_range = 1.))
        population = pg.population(prob, int(sys.argv[3]))
        population = algo.evolve(population)

    else:
        algo = pg.algorithm(pg.pso(**cfg))
        population = pg.population(prob,int(sys.argv[3]))
        population = algo.evolve(population)

    return population.champion_f[0]
```

Figura A.3: Función de evaluación *SMAC3* para la función de *Booth*.

```python
def evaluation_function(cfg):
    """
        Evaluation function
    """

    # Definition of the problem
    prob = pg.problem(xgboost_representation())

    cfg = {k : cfg[k] for k in cfg if cfg[k]}

    # Elimination of the metaheuristic, dont needed to initiate algorithms
    metaheuristic = cfg.pop("metaheuristic", None)

    # Configutation of the evaluation function to take care depending on the iteration of the alg.
    if metaheuristic == 'genetic_algorithm':
        algo = pg.algorithm(pg.sga(**cfg))
        population = pg.population(prob,int(sys.argv[3]))
        population = algo.evolve(population)

    elif metaheuristic == 'ant_bee_colony':
        algo = pg.algorithm(pg.gaco(**cfg))
        population = pg.population(prob,int(sys.argv[3]))
        population = algo.evolve(population)

    elif metaheuristic == 'simulated_annealing':
        algo = pg.algorithm(pg.simulated_annealing(Ts = 10., Tf = .1, n_T_adj = 10, n_range_adj = 10, bin_size = 10, start_range = 1.))
        population = pg.population(prob, int(sys.argv[3]))
        population = algo.evolve(population)

    else:
        algo = pg.algorithm(pg.pso(**cfg))
        population = pg.population(prob,int(sys.argv[3]))
        population = algo.evolve(population)

    return population.champion_f[0], population.champion_x[0],  population.champion_x[1],  population.champion_x[2]
```

Figura A.4: Función de evaluación *SMAC3* de *XGBoost*.

```
smac = SMAC4HPO(scenario = scenario, rng = np.random.RandomState(int(sys.argv[1])),
                tae_runner = evaluation_function)

incumbent = smac.optimize()
inc_value = evaluation_function(incumbent)


# Writing the final result to a file, to read it later
result_file_name = "seed_" +sys.argv[1] +"_" +sys.argv[2] +"_iterations_value.txt"
final_result = "Optimized value: " +str(inc_value[0]) +"\n----- Optimized XGBoost Parameters -----" +"\n - eta: " +str(inc_value[1]) +"\n - max_depth: "
                +str(int(inc_value[2])) +"\n - num_round: " +str(int(inc_value[3]))


with open(result_file_name, "w") as f:
    f.write("----- Metaheuristic configuration: -----\n" +str(incumbent))
    f.write(final_result)
```

Figura A.5: Inicialización de objetos, número de iteraciones, función de evaluación y guardado de objetos de la función de *XGBoost* en *SMAC3*.

```
import numpy as np

class booth_function:

        def fitness(self, x):
                return [(np.power(x[0]+2*x[1]-7,2)) + (np.power(2*x[0]+x[1]-5,2))]

        def get_bounds(self):
                return ([-10,-10],[10,10])
```

Figura A.6: Ejemplo de definición de una función matemática como problema de optimización en *Pygmo2*.

```
class xgboost_representation:

        def fitness(self, x):
                X, y = datasets.load_digits(return_X_y= True)

                X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7, shuffle = True, random_state = 1)

                xg_train = xgb.DMatrix(X_train, label = y_train)
                xg_test = xgb.DMatrix(X_test, label = y_test)

                # Hiperparametros definidos
                params = {}
                params['objective'] = "multi:softmax"
                params['eta'] = x[0] # [0,1]
                params['max_depth'] = int(x[1]) # [0,infinito]

                # Parametro fijo
                params['num_class'] = 10

                # Variable
                num_round = int(x[2])

                watchlist = [(xg_train, 'train'), (xg_test,'test')]
                bst = xgb.train(params, xg_train, num_round, watchlist, verbose_eval = False)

                pred = bst.predict(xg_test)
                error_rate = np.sum(pred != y_test) / y_test.shape[0]

                return ([error_rate])

        def get_bounds(self):
                return ([0,0,0], [1,6, 20])
```

Figura A.7: Definición de *XGBoost* como problema de optimización en *Pygmo2*.

```python
import pygmo as pg
from booth_function import booth_function

def main():
# Definicion del problema a optimizar
problem = pg.problem(booth_function())

# Definicion de la metaheuristica
algorithm = pg.algorithm(pg.sga(gen = 100, cr = 0.9, etac_c = 10.0, m = = 0.02, param_m = 1.0, param_s = 2,crossover =
'exponential',mutation = 'polynomial', selection = 'tournament', seed = random))

# Generacion de la poblacion
population = pg.population(problem,100)

# Evolucionamos la poblacion hasta el numero de generaciones definido
population = algorithm.evolve(population)

# Imprimimos el mejor individuo de la generacion final
print(population.champion_f)

# Imprimos los valores de los genes de ese individuo
print(population.champion_x)
```

Figura A.8: Ejemplo de optimización de la función de *Booth* utilizando un *Algoritmo Genético* en *Pygmo2*.

```python
class cnn_representation:

    def __init__(self,x_train, y_train, x_test, y_test):
        self.X_train_small = x_train
        self.X_test_small = x_test
        self.y_test_small = y_test
        self.y_train_small = y_train

        # Definimos la funcion a minimizar
    def fitness(self, x):

        model = Sequential()

        # Add model layers
        model.add(Conv2D((16 * int(x[0])), kernel_size=int(x[1]), activation='relu', input_shape=(28,28,1)))
        model.add(Conv2D((8* int(x[0])), kernel_size=int(x[1]), activation='relu'))
        model.add(Flatten())
        model.add(Dense(10, activation='softmax'))

        #compile model using accuracy to measure model performance
        model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

        model.fit(self.X_train_small, self.y_train_small, validation_data=(self.X_test_small, self.y_test_small), epochs=int(x[2]), verbose = 0)

        score = model.evaluate(self.X_test_small, self.y_test_small, verbose=0)

        return ([score[0]])

    def get_bounds(self):

        # Modification of the parameters from the neural network.
        # Kernel values are  from 1 to 5
        # Number of epochs of training goes from 1 to 5 also.

        return ([1,1,1],[5,5,5])
```

Figura A.9: Implementación de una *Red Neuronal Convolucional* en *Pygmo2*.

```python
class mlp_representation:

        # Definimos la funcion a minimizar
        def fitness(self, x):
                X, y = datasets.load_digits(return_X_y= True)
                X = preprocessing.StandardScaler().fit_transform(X)

                # Dividing dataset into train and test
                X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7,
                                                        shuffle = True, random_state = 1)

                mlp = MLPClassifier(hidden_layer_sizes=(int(x[0]),), activation='logistic', alpha=1e-4,
                                    solver='lbfgs', tol=1e-4, random_state=1,
                                    learning_rate_init=x[2], verbose=False, max_iter = int(x[1]))

                mlp.fit(X_train,y_train)

                predictions = mlp.predict(X_test)

                return ([1 - accuracy_score(y_test, predictions)])

        def get_bounds(self):
                return ([6,250,0.00001], [64,10000, 0.1])
```

Figura A.10: Definición del MLP como problema de *pygmo2*.